A Multiview Visualisation Architecture for Open Distributed Systems

Petra Oldengarm, Aart van Halteren
KPN Research, P.O. Box 15000
9700 CD Groningen, The Netherlands
{P.Oldengarm, A.T.vanHalteren}@research.kpn.com

Abstract

Program visualisation is an attractive way for understanding collaboration structures of complex distributed systems. By using the concepts of the Open Distributed Processing - Reference Model (ODP-RM) as entities for visualisation, a multiview visualisation architecture is presented, which provides a large degree of flexibility in visualising the actions of an ODP system. The architecture has been implemented for visualising CORBA systems resulting in a visualisation tool called OBVIouS.

Introduction

The number of distributed systems is growing continuously. Also downsizing mainframe applications to client-server applications has become a popular practice. In fact, new technologies for developing distributed applications such as CORBA, Java and ActiveX are turning the Internet into a large distributed object system. These developments provide enterprises with new opportunities, but also with the additional challenges of managing new paradigms for developing and deploying services.

Developing distributed systems is a complex task. Tools and modelling languages such as the Unified Modelling Language (UML) are helpful, but have no explicit way for expressing distribution of objects over the network. Therefore a means to give a clear understanding of the collaboration and distribution of objects is an important, [boilerplate] ogram visualisation can help gain insight into how a distributed system works and could be used to discover or prevent potential bottlenecks. In addition, program visualisation can expose design errors in the distribution strategy of an object system.

Based on the concepts defined in ODP-RM [ODP-1 1995] [ODP-2 1995] [ODP-3 1995] [ODP-4 1995] we present an architecture for visualising a wide variety of open distributed systems. From this architecture we describe a flexible and easy to configure tool for program visualisation of distributed object systems. The visualisation architecture has been applied to and implemented for CORBA systems [OMG 1997].

Program Visualisation

Graphical visualisations are useful and powerful for understanding complex tasks. These visualisations can be divided into three area's [Tomas 1994]:

- Scientific Visualisation: using graphical representations of data to gain insight in the structure of that data.
- Visual Programming: specifying a program in a two-dimensional graphical form.
- Program Visualisation: using visualisation to gain insight in the behaviour of a program.

We focus on Program Visualisation, which provides insight in the operation of a program and can be used for debugging, performance evaluation or educational purposes. Program Visualisation of distributed systems is a relatively new area of research. However, much research has been done towards the visualisation of parallel systems. The problems that arise in the visualisation of parallel systems, are also present for the visualisation of distributed systems.

Examples of tools that perform parallel program visualisation are described in [Bode 1993] and [Joyce 1987]. Most visualisation tools divide the visualisation process into four steps [Krae 1993]:

1. data collection: collect all available information from the system that is visualised.

System	Advantages	Disadvantages
WALTER and ELVIN [Bond 1994]	• Separation of concerns: the event logging part and the visualisation part are separated.	• DCE is used, which is not an object oriented environment. • The event logging calls have to be added to the client and server code by the programmer of the distributed system.
ObjectMonitor [Brüh 1996]	• A CORBA system is monitored.	• A complicated method is used to retrieve information from the system (no separation of concerns). • The programmer of the distributed system has to add code to the client and server code.
Management system [Brun 1996]	• A CORBA system is monitored. • Separation of concerns: the requests are filtered and generate events, which are handled to provide the visualisation.	• The programmer has to add *management information* during the implementation phase of the distributed system. • The programmer has to add a lot of extra code in all stages of the implementation phase.

Table 1: Comparison between visualisation tools for distributed systems

2. data analysis: decide which information has to be visualised.
3. storage of data: store the visualisation data for later use.
4. display: display the visualisation data.

We use these steps in the design and implementation of our tool.

A few publications on visualisation of open distributed systems have been found. We have compared the efforts of Bond [Bond 1994], Brühan [Brüh 1996] and Brunne [Brun 1996]. Table 1 gives an overview of the advantages and disadvantages of the various tools. The tools described in the table have too many disadvantages to be useful for us. Therefore we have developed our own visualisation tool eliminating several of these disadvantages. The design and implementation issues of this tool are described in the remainder of this paper.

Visualisation with ODP-RM

As a target language for describing the actions that happen in the distributed system that is visualised, we use ODP-RM. The five viewpoints of ODP-RM (the *enterprise*, *information*, *computational*, *engineering* and *technology* viewpoint) provide a nice starting point to describe a system for different groups of users. Each viewpoint emphasises different aspects of the system. All viewpoints have their language, which is a set of concepts for describing the system. ODP-RM provides a way to abstract from details that are not relevant in a certain view on the system.

We apply the ODP concepts to CORBA systems, since CORBA can be interpreted as an *instance* of the *class of ODP systems*. To describe how a CORBA system is visualised, we have to relate ODP-RM concepts to aspects in CORBA. In principal all ODP viewpoint languages are candidates for describing the aspects of a CORBA system to visualise.

CORBA implementations have the following characteristics. Clients and servers are (part of) processes that run on machines with certain operating systems. Those clients and servers communicate with each other through TCP/IP connections (when the Internet Inter ORB Protocol (IIOP) is used) and they are programmed in object oriented programming languages, like C++ and Java, or in conventional languages like C and COBOL. They consist of a set of objects written in any of these languages. These characteristics find a detailed match to the concepts of the engineering language of ODP-RM. The engineering language is the most elaborate viewpoint language, with which we can describe a large set of concepts without losing generality. This is in contrast with the other viewpoint languages. With the enterprise language we can only describe a small part of the concepts that are present, because the enterprise viewpoint abstracts from too many issues. The technology language is limited in size and not very generic, so it does not provide a good base for describing what happens in a CORBA system. The computational and information language do not explicitly deal with the physical distribution of objects, so they are not suitable for describing all aspects involved in the visualisation of CORBA systems. Therefore we have chosen the

engineering language for describing what happens in a CORBA system.

For the concepts defined in the engineering viewpoint language we give a mapping to aspects that can be identified in the CORBA system.

- A *node* in ODP-RM is a physical abstraction of a (physical) computing system. So the node is the *machine* (or *host*) on which objects in a CORBA system runs.
- Every node in an ODP system is under control of a *nucleus*. A nucleus is an abstraction of the *operating system*. A CORBA system can run under UNIX or Windows NT, and many other operating systems.
- A *capsule* can be compared to an operating system process with its own address space. An important feature of a capsule is that when it fails, it should not affect other capsules. On every node a capsule has a unique process id.
- A *capsule manager* manages the engineering objects in a capsule. There is no generic mapping from a capsule manager to an aspect in a CORBA system. It is possible to implement a capsule manager in a CORBA system with a *factory object*, or an API to the operating system that organises the instantiation and deletion of objects within a capsule. In this way it controls the interactions within a capsule. This is a design specific issue.
- A capsule can be decomposed into parts that perform a certain function. These parts are called *clusters* in ODP-RM. Objects are grouped into clusters to reduce overhead to manage them individually. A designer can e.g. define a cluster as a group of C++ objects and object implementations. This is also a design specific issue.
- Clusters are controlled by *cluster managers*, which perform actions on the clusters like instantiation and deletion. There is no generic equivalent of these managers in a CORBA system. They can be implemented by *administration objects* that keep track of the objects in a cluster. Therefore, cluster managers are design specific.
- The objects in a cluster are the basic elements of the system and they are called *basic engineering objects* (BEOs). In CORBA these objects are instances of single classes (e.g. C++, Java, Smalltalk classes, or object implementations).
- Objects in different capsules communicate through *channels*. In a CORBA system using IIOP a channel is established as a *TCP/IP connection* between two processes.

Table 2 summarises the relationships that we described in the previous paragraph.

ODP-RM	CORBA
Node	Machine
Nucleus	Operating system
Capsule	Operating system process
Capsule Manager	*design specific*
Cluster	Set of objects in same process
Cluster Manager	*design specific*
Basic Engineering Object	Object instance
Channel	TCP/IP Connection

Table 2: Relationship between CORBA and ODP

A generic method for visualising open distributed systems

Visualising open distributed software systems, requires a process of four phases:

1. The *event collection* phase, during which the available information is retrieved from the system.
2. The *event processing* phase, during which the collected events are ordered and translated into events that are used in the visualisation.
3. The *storage* phase, during which the visualisation events are stored in a queue, or a log file.
4. The *display* phase, during which the visualisation events are translated into graphical primitives (i.e. shape, colour, position, etc.) and displayed.

An important part of the design of our tool is to create a set of events that gives a full description of the actions that are performed in the system that is visualised. We distinguish two types of events:

- *ORB events*: events that are retrieved during the event collection phase of the tool.
- *Visualisation events*: events that are displayed during the display phase. These events are the output of the event processing phase.

Figure 1 shows the in- and output of each of the four phases in the visualisation process.

In the following paragraphs we give a description of the ORB and Visualisation events.

The ORB events are collection of all the available information from the distributed system that is visualised.

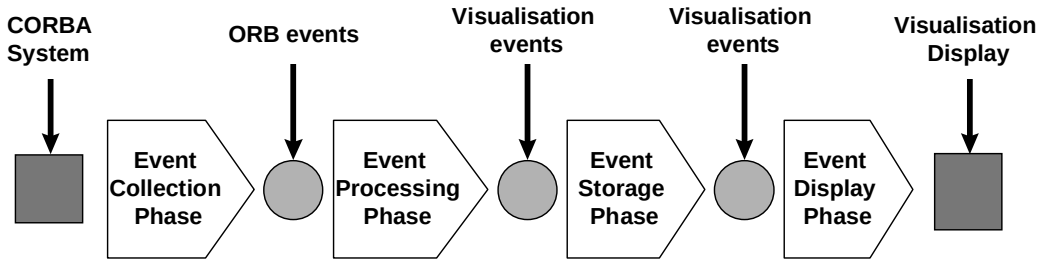

Figure 1: The visualisation process

Table 2 gives a full description of the information that can be obtained from the system. We have constructed three types of ORB events. The common characteristics of these events are defined in the superclass *ORBaction.*

Figure 2 depicts the structure of the ORB actions in UML notation .

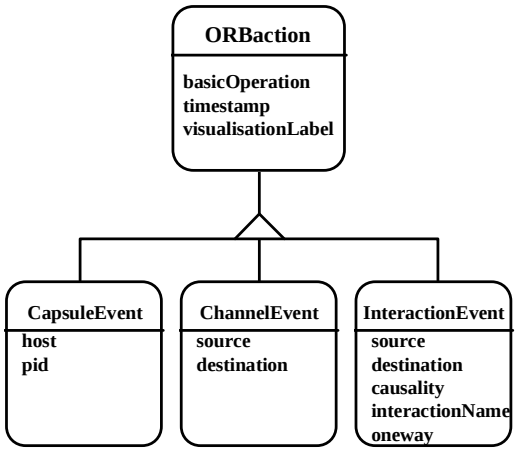

Figure 2: Schematic overview of ORB events

Every ORB action has three basic attributes:

- `basicOperation`: every action has an instantiate or a destroy tag.
- `timestamp`: the time at which an event happens.
- `visualisationLabel`: a label that is used by the graphical user interface to give a certain object a pre-defined appearance. If this label is not set, the object gets a default appearance.

In addition to these attributes, the `CapsuleEvent` has a `host` and a `pid` attribute that give a unique identification for the capsule. This kind of event is generated when a capsule is created or destroyed. The `ChannelEvent` has attributes to determine what the `source` and `destination` of the channel is. This kind of event is generated when a channel is created or destroyed. The `InteractionEvent` has attributes for the `source` and `destination` of the interaction, the `causality` (i.e. whether the operation is a request or a reply), the name of the interaction (`interactionName`) and an attribute to determine whether the interaction is a `oneway` request.

The set of Visualisation events is larger than the set of ORB events. It must be possible to generate events for every viewpoint of ODP-RM. The set that we have created is not yet complete, and will be extended in the future, based on the experiences of the users of our tool. All the ORB events are also Visualisation events. At this moment we have designed two extra Visualisation events that can be generated (see Figure 3).

Like the ORB events, the `ClusterEvent` and the `ActorEvent` inherit all the attributes from `ORBaction`. The `ClusterEvent` has attributes to determine the identity of the capsule that the cluster belongs to (`host` and `pid`). The `clusterName` gives a unique name for the cluster within the capsule. The `ActorEvent` has an attribute that gives the `name` of the actor.

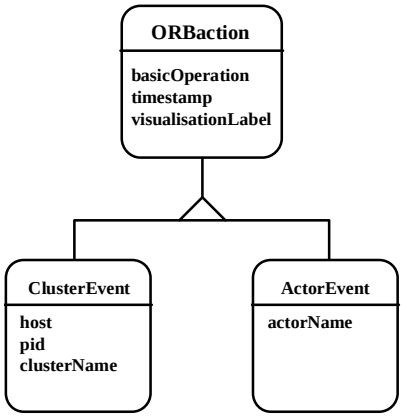

Figure 3: Schematic overview of a part of the Visualisation events

To use the ORB and Visualisation events to visualise a CORBA system, the events have to be translated into Interface Definition Language (IDL) type definitions.

Design of the visualisation tool

In this section the design of a tool for visualising CORBA systems is described. The tool is called OBVIouS (OBject VIsualisation System). We first describe the target group of users. Then we give an analysis of the requirements. The section ends with a description of the architecture of the tool.

Target group of users

The purpose of the visualisation depends on the target group of users:

- Researchers and developers of distributed systems could use program visualisation for debugging purposes and for gaining insight in the information flow of the program. With the tool they could also detect potential bottlenecks in the communication.
- Managers could use program visualisation to get a high-level understanding of the distributed system.
- System designers and architects could use the tool to see if the CORBA system has been implemented according to their design.
- End users of the distributed system could use program visualisation to get an idea of the things that happen in the system.

Requirements for a visualisation tool

From interviews with developers of distributed systems we have composed the following set of system requirements, which apply to our tool:

- The efforts on the software developer for visualising an application should be minimal. This implies that the developer shouldn't have to add much code to the application for performing the visualisation.
- The tool must be capable of visualising multiple views on the same system. We define these views using the viewpoints from the Open Distributed Processing - Reference Model (ODP-RM).
- The view on the system must be defined by a visualisation template in an easy to use language.
- The appearance of the entities that are visualised must be easy to configure.
- Visualisation is performed while the distributed system is running.
- Detailed information must be displayed when demanded (e.g. the computing system an object is running on).

Tool architecture

The design of the tool incorporates the four phases discussed before and handles each phase separately. The architecture is shown in Figure 4.

The generation of ORB events is performed in *Filters*, that push the events into an *EventChannel*, for which we use the CORBA Event Service. In this phase of the process the question *which actions are performed by the CORBA system* is answered.

The EventChannel pushes on its turn the ORB events into a server that performs the event processing. The ORB events are translated into Visualisation events by the *Rule Engine* of this server. In this phase it has to be determined *what has to be visualised*.

The processed events are stored in an event queue and a log file during the third phase.

We have designed a graphical user interface that retrieves the visualisation events from the event queue, adds display information for the visualisation and displays the information. In this phase an important question is *what appearance should the events have*.

Implementation of the tool

The OBVIouS tool has been implemented at KPN Research. This section discusses the implementation issues involved.

Event collection

Many of the distributed systems at KPN Research have been built with Orbix, therefore the current implementation of the tool is targeted for Orbix.

The ORB events that we want to collect from the system that has to be visualised have to be generated at some point in that system. Orbix has a filter mechanism which gives the possibility to add code at certain *filter points* to retrieve information about a certain request. We have built filter objects that use this mechanism. The information that passes these filter objects gives all the information we need to generate all types of ORB events.

When the filter objects are instantiated or deleted, we know that a capsule has been created or destroyed. When a TCP/IP connection opened or closed, we let Orbix notify the filters about it. In this way we know exactly when to generate a channel event.

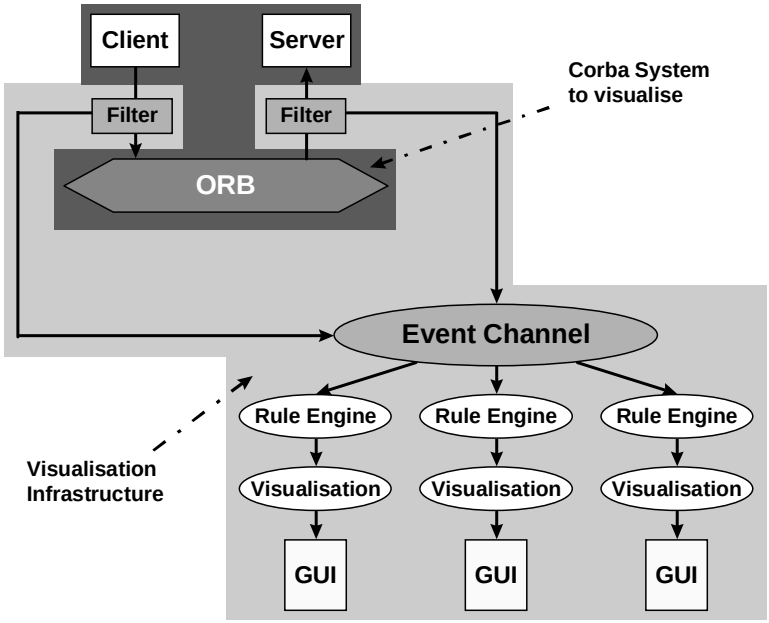

Figure 4: Architecture of the OBVIouS tool

The filter points in the filter object provide information about requests that are made between a client and a server. We use the filter points to observe in-requests, out-requests, in-replies and out-replies. When a certain request/reply is made, we generate an interaction event.

For the distribution of events we use OrbixTalk, an implementation of the CORBA Event Service. OrbixTalk handles the transport of the events through the event channel to interested server objects (see Figure 4).

To use the filter objects the developer only has to add one line of code to each capsule that contains objects that must be visualised. The code that has to be added performs the instantiation of the filter objects.

Event processing and storage

To process the events, we have designed an Event Listener server that translates the ORB events into Visualisation events. However, this translation depends on the design of the CORBA system and on the kind of view that has to be created. The translation is not hard coded into the Event Listener server itself, but is based on a script language. Without recompilation the Event Listener server can take another script that performs another translation of the events, thus creating another view on the system. The script serves as a *rule engine* in the visualisation process (see Figure 4). In our implementation we use Perl as the script language.

The ORB events that are pushed to the Event Listener server are stored in a queue. An event thread processes these events by pushing them into the Perl script and putting the result that is returned in another queue. A problem here is the ordering of the ORB events. The events don't necessarily arrive in the correct order at the Event Listener server. At this moment it is only possible to make a partial ordering on the interaction events (requests are performed in the order: instantiate-request, destroy-request, instantiate-reply, destroy-reply). For the absolute ordering we need to use the timestamps that we send along with the events. At this moment we have ignored this problem and assumed that the events do arrive in the correct order.

Besides the translation of events, the Event Listener server provides an interface which gives the possibility to retrieve the next available Visualisation event from the event queue. The IDL specification of this interface is given below.

```
interface EventListener_IF {
  VisualisationEvent getNextEvent();
};
```

A VisualisationEvent is one of the Visualisation events that we have described in the previous section.

To make different views on the system the developer of the CORBA system has to write a Perl script for every view. To run the different views simultaneously different instances of the Event Listener should be created with a unique name. These instances can be created with the

user interface that we have designed. The user interface is explained in more detail in the following section.

Display of the visualisation events

For the display of the Visualisation events we have designed a graphical user interface in Java. This interface retrieves events from the Event Listener interface and displays them on screen. Each view, however, has its own graphical primitives for displaying the information, i.e. its own colours, shapes, etc. Therefore we have constructed a configuration file, which gives the possibility to add this kind of information to the user interface.

The configuration file must be written according a specified grammar. For this grammar we have constructed a scanner and a parser using the scanner generator JLex [Berk 1997] and the parser generator CUP [Hudson 1996]. This gives the possibility to easily adapt the configuration grammar to new requirements.

In the current version of the grammar, we must specify the name of the view, the relative size of it and the background as a colour or as an image. We must specify how capsules look and where they should be placed on the screen, initially. A capsule can be drawn as a rectangle, a circle, or as an image that is specified. From an interaction and a channel, the colour must be specified. The colour can be specified as one of 13 predefined colours or with RGB components.

When a new Visualisation event arrives at the user interface, it's appearance is determined by the configuration settings of the configuration file. For this the visualisationLabel attribute is used. If no configuration is found, the event is displayed with a default appearance.

Conclusions and recommendations

In this paper we present an architecture for visualising open distributed systems. We have developed a generic method for visualising distributed applications and applied this method to CORBA systems. This resulted in a visualisation tool called OBVIouS.

In our approach on visualising open distributed systems, we have taken the concepts from ODP-RM and identified concepts suitable for visualisation. The architecture we present manipulates this set of visualisation entities in several phases. These phases have proven to be necessary, in order to allow a large degree of flexibility in what is visualised and the appearance of visualisation entities.

In understanding the key features of the distributed system, ODP-RM provides a good reference. We have defined a mapping from entities in a CORBA system to ODP-RM concepts. This gives us the possibility to describe the distributed system with ODP-RM and then translate this description into events that occur in the distributed system.

With the generic method we constructed a highly configurable tool, called OBVIouS, that can visualise CORBA systems while they are operating. Our tool gives the possibility to produce different views on the system simultaneously. The programming efforts for the developer of the CORBA system are kept to a minimum. Only one line of code has to be added to each operating system process to enable visualisation. This single line of code instantiates so-called filter objects that generate the ORB events. We have applied the CORBA Event Service for receiving and distributing the events. This service allows a loose coupling of the visualisation tool from the system that has to be visualised. By means of a Perl script the events generated from the CORBA system can be filtered out, aggregated or transformed into visualisation events. A configuration file is used to instruct a Java applet on the appearance and representation of visualisation events.

Currently, our visualisation tool has been implemented with Orbix. In our tool we use the Orbix specific filter mechanism, which gives us the possibility to generate events when requests are made. The CORBA 2.2 standard defines interceptors that have the same functionality as the filters of Orbix. This implies that distributed object systems implemented using other ORB implementations can be visualised with our tool as soon as interceptors are widely available.

We recommend that ORB implementations provide a standard set of events that can be generated in a CORBA system. This implies that an application developer doesn't have to add any code to enable visualisation. The ODP-RM provides a nice set of concepts that can be used to define a standard set of visualisation events. This set could be further refined and standardised by the Object Management Group (OMG).

In addition, the ORB could provide a management interface to configure the level of detail of the visualisation events. This implies that during a visualisation session, it can be decided that some of the events don't have to be generated , because they are never used. This improves the performance of the system.

Literature References

[Berk 1997] E. Berk, '*Jlex: A lexical analyser generator for Java ™*', Department of Computer Science, Princeton University, http://www.cs.princeton.edu/~appel/modern/java/Jlex/manual.html, valid in December 1997, 1997.

[Bode 1993] A. Bode and P. Braun, '*Monitoring and Visualisation in TOPSYS*', Performance Measurement and Visualisation of Parallel Systems, Proceedings of the workshop, pp 97-118, Elsevier Science Publishers B.V., 1993.

[Bond 1994] A. Bond and D. Arnold, '*Visualising Service Interaction in an Open Distributed System*', proceedings of IEEE Workshop on Services for Distributed and Networked Environments, pp 19-25, 1994.

[Brüh 1996] M. Brühan and S. Ruppert, '*ObjectMonitor: Monitoring von CORBA-Anwendungen*', Fachhochschule Wiesbaden, http://wwwvs.informatik.fh-wiesbaden.de/programme/objmon-visco/, valid in November 1997, 1996.

[Brun 1996] H. Brunne and T. Usländer, '*Design of a Monitoring System for CORBA-based Applications*', Trends in Distributed Systems '96, Industrial and Short Paper Proceedings, Workshop RWTH Aachen, pp 52-66, 1996.

[Hudson 1996] S.E. Hudson, '*LALR Parser Generator for Java ™ Cup*', Graphics Visualisation and Usability Centre, Georgia Institute of Technology, http://www.cc.gatech.edu/gvu/people/Faculty/hudson/java_cup/manual.v0.9e.html, valid in December 1997, 1996.

[Joyce 1987] J. Joyce, G. Lomow, K. Slind and B. Unger, '*Monitoring Distributed Systems*', ACM Transactions on Computer Systems, vol. 5, no. 2, pp 121-150, 1987.

[Krae 1993] E. Kraemer and J.T. Stasko, '*The visualisation of Parallel Systems: An Overview*', Journal of Parallel and Distributed Computing, vol. 18, no. 2 pp 105-17, 1993.

[Lamp 1978] L. Lamport, '*Time, clocks and the ordering of events in a distributed system*', Communications of the ACM, vol. 21, no. 7, pp 558-565, 1978.

[ODP-1 1995] ITU/ISO, Open Distributed Processing - Reference Model, '*Part 1: Overview*', International Standard 10746-3, ITU-T Recommendation X.903, 1995.

[ODP-2 1995] ITU/ISO, Open Distributed Processing - Reference Model, '*Part 2: Foundations*', International Standard 10746-3, ITU-T Recommendation X.903, 1995.

[ODP-3 1995] ITU/ISO, Open Distributed Processing - Reference Model, '*Part 3: Architecture*', International Standard 10746-3, ITU-T Recommendation X.903, 1995.

[ODP-4 1995] ITU/ISO, Open Distributed Processing - Reference Model, '*Part 4: Architectural Semantics*', International Standard 10746-3, ITU-T Recommendation X.903, 1995.

[OMG 1997] Object Management Group, '*The Common Object Request Broker: Architecture and Specification, Revision 2.0*', OMG document 97.2.25, 1997.

[Rup 1996] S. Ruppert, M. Brühan and Oliver Bildesheim, '*VISCO 1.0 - VISualisation of Corba Objects*', Fachhochschule Wiesbaden, http://wwwvs.informatik.fh-wiesbaden.de/programme/objmon-visco/, valid in November 1997, 1996.